CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

5042:TR:82

FORMAL SPECIFICATION OF CONCURRENT SYSTEMS

by

Marina C. Chen
and
Carver A. Mead

Presented at
USC Workshop on VLSI and Modern Signal Processing
November 1982

The research described in this report is sponsored by
System Development Foundation and its initial phase was sponsored by
Defense Advanced Research Projects Agency ARPA Order #3771 and monitored by
Office of Naval Research Contract #N00014-79-C-0597

Preface

This report contains the first published account of a new framework for the specification and analysis of concurrent systems. This framework is based upon a formal system with precisely defined semantics. This formal system can be viewed as a new approach to programming language. The formalism has been shown to apply to any level of system definition. It has been used to specify transistor circuits, logic gates, arithmetic units, sequential processors, and emsembles of large number of communicating processes. The most important attribute of the framework is its ability to abstract from one level of specification to the next. The two papers in this report were given at the USC Workshop on VLSI and Modern Signal Processing, November, 1982, and appear in the Proceedings of that conference. They are reprinted here for distribution to a wider audience.

VLSI, SIGNAL PROCESSING, and FORMAL SEMANTICS

Carver A. Mead

Signal processing has historically been the only widely practiced form of computing based upon a formal semantics. The properties of linear systems have allowed the behavior of a computational module to be defined independent of the input data upon which it will act. Although it is so familiar we all take it for granted, the linear transfer function is perhaps the most powerful engineering abstraction ever invented. Components can be composed by merely multiplying their transforms. It is commonplace for exceedingly complex filtering computations to be designed, implemented, and perform their function precisely as specified without a single redesign.

Contrast this situation with the common practice of computer programming. Specifications are vague. Most important aspects of an application are discovered in the process of implementation. Interfaces between modules are never precisely specified and cause enormous difficulty. Although advances in the art have been made in recent years through better modularity and more precise specification of interfaces, one astonishing fact remains – the only precise definition of the function of a system is the code which implements it! This situation is glorified with the title "Operational Semantics" in the computer sicence world, meaning "The Code Does Whatever It Does".

In the past when the world was viewed as one lone sequential process, it was perhaps possible to delude oneself into believing that a program was its own best definition. In today's world, algorithms are mapped onto silicon, distributed in space and time. The sequential way is no longer the only way – in most cases it is not even a viable contender. New approaches to algorithms are emerging daily, many of the best are represented at this conference. It is essential to be able to compare the area, time, and power required by several implementations of the same function. The ability to precisely specify the function performed by a system thus assumes far greater importance for the future than it has in the past.

In spite of the lack of formal specification and analysis, a small group of highly skilled professionals can build and maintain exceedingly ambitious software projects. It is instructive to observe how they do it. Universally, they take a hierarchical divide and conquor approach. The application is broken down into a small number of component modules. These are given names which clearly indicate their function. Each of these modules is further broken into sub-modules which are also carefully named. The process continues until the lowest level modues can be implemented directly. Complexity management is done through reasoning about a set of abstractions. The definition of each abstract function is carried by its name.

Since the earliest days of programming, attempts have been made to ascribe a precise definition to the abstract function performed by a program, apart from its

implementation. If each part of a program performs an operation which is described by a mathematical function, the entire program can be viewed as the composition of its component functions. Powerful properties of mathematical functions can be brought to bear. This approach has evolved into the so-called applicative languages such as pure LISP, Bacus's FP, etc. A precise abstraction can be given for non-linear functions. However, one difficulty is encountered. There is no way to represent state. A bank account would be described by the entire history of deposits and withdrawals rather than by a simple balance. While functional notation provides an excellent mechanism for abstracting function, it does not allow for the abstraction of history! This inability to deal with state has prevented the purely functional or applicative approach from being widely used.

From a signal processing perspective, this state of affairs is incomprehensible. After all, a recursive filter retains traces of its input from an indefinite period into the past. The state stored in the delay elements of the filter is just the right abstraction of that history to give the filter its well defined transfer function. This point is not a trivial one – it clearly demonstrates that the presence of state does not, in itself, prevent a mathematical abstraction of the function.

We are led to ask whether we can, in some sense, have the best of both worlds – the power of general non-linear functions together with the linear systems' ability to abstract an element with state. The answer is yes, and details are given in Marina Chen's PhD thesis (CalTech, 1982). A brief account is presented for the first time in our companion paper in this conference. The success of this project is a direct result of its determination to understand and formalize real engineering practice used in the design of real systems. The key to understanding the dilemma came from signal processing. The Z-transform allows us to talk about events ordered in time. Since state is an abstraction of time history, no semantics can handle it properly if relations in time cannot be described. The purely functional approach ignores time completely, and thus cannot represent a system with state. In our treatment we combine the generality of the functional approach with the explicit time relationships necessary to represent state.

Armed with a formalism for describing and analysing general non-linear functions in a system with state, we are in a position to describe the broad range of real machinery which can be implemented in the VLSI medium. We can truly abstract from circuit and gate level to high level functions. I firmly believe that this approach will form the basis for a unified semantics of computation. However we must not underestimate the work left to be done.

Since the underlying space-time representation can describe any arbitrary system, it is in some sense like a blank piece of paper. It is as if we have just discovered differential equations, but have not as yet developed solution techniques. Of all possible classes of systems, only the familiar linear ones have evolved a sophistocated calculus. Using the space-time formalism for a wider range of algorithms is an important task for the future. I believe the effort will be well rewarded. For the first time we can discuss systems of all types, at all levels, in a common language.

Concurrent Algorithms as Space-time Recursion Equations*

13 October 1982
Marina C. Chen and Carver A. Mead†

Introduction

It is by now well recognized that VLSI technology has brought about a medium which allows the realization of orders of magnitude more computing elements per unit cost. The more significant contribution of VLSI to Computer Science will be in the utilization of many hundreds or thousands of these elements concurrently to achieve a given computation. It is clear from existence proofs of such innovative designs as systolic arrays [KUNG & LEISERSON80], tree machine algorithms [BROWNING80], computational arrays [JOHNSSON ET AL.81], wavefront arrays [KUNG,S.Y.80], etc. that vast performance improvements can be achieved if the design of so-called "high-level" algorithms is released from the one dimensional world of a sequential process, and the cost of communications in space as well as cost of computation in time is taken into consideration [SUTHERLAND & MEAD77]. While this higher dimensional design space provides a great playground for innovative algorithm design, it also introduces pitfalls unapprehended by those accustomed to the world of a single sequential process. Verification of algorithms becomes much more crucial in system designs because debugging concurrent programs can very easily become an exponentially complicated task in this rich space. The real difficulty lies in the high degree of complexity of concurrent systems. The well-known hierarchical approach can be used to manage the design complexity for such systems. A system is broken down into

*This work is sponsored by System Development Foundation and its initial phase by Defense Advanced Research Projects Agency ARPA Order #3771 and monitored by Office of Naval Research Contract #N00014-79-C-0597. One of us (M.C.) was supported by an IBM Doctoral Fellowship

†Computer Science Department, California Institute of Technology, Pasadena, California.

successive levels of sub-systems until each is of a manageable complexity. The effectiveness of this approach relies on two basic tools: A design and verification methodology for each level and an abstraction mechanism to go from one level to the next. The latter is crucially important, for without it the consistency of the whole system is imperiled.

In this paper, we describe a methodology and a single notation for the specification and verification of synchronous and self-timed concurrent systems ranging from the level of transistors to communicating processes. The uniform treatment of these systems results in a powerful abstraction mechanism which allows management of system complexity.

Traditionally, due to the assumption that the cost of accessing variables in memory is the same regardless of their locations, sequential algorithms ignore the spatial relationships of variables. In addition, the steps of a computation have not been explicitly expressed as a function of time, but are rather implied by programming constructs. Languages that cannot express the spatial relationships of variables cannot take into account the most important aspect in the design of a concurrent algorithm, i.e. ensuring locality of communications, taking advantage of the interplay of variables in space (in practice up to 3 dimensions) to achieve higher performance. The implicit "time" causes programming languages to suffer either from not being able to abstract the history of computation (e.g. in applicative and data-flow languages [KAHN74, BAKUS78]), or not being able to abstract computation in a clean functional form (e.g. in assignment-based languages). Here we choose to make "time" an explicit parameter of computation. We call our representation of computation a "Space-time Algorithm".

In [CHEN82], CRYSTAL (Concurrent Representation of Your Space Time ALgorithm), a notation for concurrent programming is proposed. The fixed-point approach [SCOTT & STRACHEY71] is used for characterizing the semantics. Within this framework, a program is expressed as a set of systems of recursion equations. Unknowns of the equations are data expressed as functions from the space-time domain to the value domain.

For a deterministic concurrent system, such as a systolic array, a single system of equations results, and the semantics of such a system is defined as the least solution of the equations. The semantics of concurrent systems in general can be characterized as the corresponding set of solutions of the set of systems of equations. In this paper, we concentrate on deterministic concurrent systems at the communicating sequential processes level. We will first present briefly considerations that are generic to all systems, i.e., the underlying model of computation, the representation, and the mathematical semantics of the systems. Various inductive techniques (see for example [MANNA74]) used in verifying recursive programs can be directly applied in verifying space-time algorithms and proving their properties. We demonstrate this framework by presenting both the synchronous and self-timed version of the matrix multiplication on systolic arrays [Kung & Leiserson80] with its proof of correctness. The notion of wavefront is especially important in this class of computations. We define the "phase" of a computation wave in a way that is analogous to the wave in physical world. The set of all possible "phases" can be formalized as a well-founded set, upon which the inductive proof is based.

Model of Computation

The model consists of an ensemble of *sequential processes* each of which has its own local state and ports for communicating with other processes. Depending on the level of system concerned, these processes can be as simple as a single transistor or as fancy as a conventional von Neumann type machine. A sequential process consists of a function that maps from inputs and current-states to outputs and next-states. Such a function uniquely defines a sequential process. It is the generator of the output sequence and state with given initial state and an input sequence. The state captures the semantic abstraction of the history. No assertion about the process can depend upon history in a way not captured by the state. A single invocation of the function i) evaluates the function, ii)updates the state and outputs, iii)increments the process's "time", all as an atomic event.

In a particular process, "time" is a measure of how many invocations have occurred,

and "space" is where the process is located. "Time" is a property local to each process. Note that state is explicitly represented, a function is not defined from the history of inputs to outputs as in the applicative and data flow model of computation. Communications among processes in space are specified by identifying inputs of one process with outputs of other processes in the space-time domain. Also note that a transition from one invocation to the next within a sequential process can be viewed as a communication in the time domain (fixed in space).

The "slicing" of a sequential process into a sequence of functions is done at the communication with the external world. Inputs from several different processes which are aligned in time and used as arguments to a single function are considered as one external input event, i.e. one invocation. Within the same slice, no side-effects are allowed, i.e. each slice is strictly functional. We enforce this discipline by using a purely applicative programming notation (like pure Lisp and Backus's FP notation) to implement atomic functions, which cannot be further sliced either in space or in time. Any higher level system is constructed by composing atomic functions and other existing systems using recursion equations. The resulting system is always transformed into a function from inputs and current-states to outputs and next-states, i.e. a sequential process. It is often the case that once the system is implemented, a sequence of inputs can be conveniently considered as a set of inputs at the next level up. Although internal state is used as part of the implementation, the outputs can be expressed as a function of such a sequence of inputs without refering to the state. In such a case we abstract the process as an applicative function and it can, once again, be treated as if it were atomic. In real-time systems, this kind of abstraction is not possible since the sequence of inputs cannot be treated as a static input, making explicit state still necessary.

Thus space-time algorithms are either purely applicative programs or recursion equations. Note that in this way, states can be expressed without side-effects. The change from viewing an applicative system as the universe to using it only for an atom is the key

to the applicability of our framework to real systems. The applicative model of computing suffers one major drawback in not being able to retain the result of a computation so that it can be used in a different place or at a later time. The data flow model is a remedy for this problem only in space. The essential ability to use a result in several places is captured by the data-flow equations devised by Kahn [KAHN74]. Unfortunately, this model still lacks the essential capability of capturing the state, the result in the time domain. This fact is manifested in the proliferation of assignment-based data-flow languages [ACKERMAN, W.B.]. The elegance of data-flow equations cannot help the implementation of real world systems where state is necessary. The space-time recursion equations using a purely applicative language can be applied to real world problems*. This insight is the most important contribution of our work to computer programming. We thus retain the elegance and formal cleanness of functional application together with the essential ability to abstract history into a compact form.

Representation

Let $\mathbf{x} = (x_1, x_2, \ldots, x_m)$ denote the inputs and current-states of a process, where $x_i \in D_i$ for $i = 1, 2, \ldots, m$. Here D_i is the domain* where the input or current-state i assumes its value. We define a function f which maps the vector of inputs and current states to the vector of outputs and next states:

$$f : D^m \rightarrow D^n,$$
$$f = (\lambda \mathbf{x}.f_1, \lambda \mathbf{x}.f_2, \ldots, \lambda \mathbf{x}.f_n) \qquad (1)$$

where n is the total number of outputs and states.

Each component, $\lambda \mathbf{x}.f_i$, of such a function is an element of $[D^m \rightarrow D]$. These functions must be monotonic (see for example [MANNA 74]) over D^m.

In order to capture the notion of flow of the data and the structure of these data, we define *data streams*. Each "stream" of data is represented by a function from the

*The drawback of applicative languages is best captured in a quote by Perlis "Purely applicative languages are poorly applicable ." [PERLIS 82]

*Technically, domains are not sets but *complete lattices* with *approximation ordering*. Please refer to [MANNA74] and [SCOTT & STRACHEY71].

space-time domain $\mathcal{V}$ to the value domain $\mathcal{D}$. We next define *structured processes* which define the location of the processes making up the ensemble, as a function in the domain $[\mathcal{V} \rightarrow [\mathcal{D}^m \rightarrow \mathcal{D}]]$. This function is defined by cases for different process types in the space. Cases are specified in the notation of [DIJKSTRA 76]. The complexity of this definition reflects the heterogeneity of the process types.

The relationship among the structured input and output data streams and structured processes and functions are defined in a point-wise manner. In the space-time domain, an output is the result of the application of the function at that point to input data streams at that point. Through connections, the input (current-state) streams of one function are identified with output (next-state) streams of other functions, or with initial/boundary values. We therefore define *structured connections* also as a function in the domain $[\mathcal{V} \rightarrow [\mathcal{D}^m \rightarrow \mathcal{D}]]$. The description of this function reflects the regularity of the connections. By substituting the equations of structured connections into those of structured processes, we obtain a system of recursion equations that define output streams in terms of output streams and initial/boundary conditions.

An obvious restriction on these recursion relations is that the time components can only increase by one unit at a time, i.e. an argument presented to the input of a function at its "time" t will affect the value of that function which appears at its output at its "time" $t+1$.

In general, an input stream at a given point in the space-time domain can connect to an output stream at any other point in space. In specific cases, such as when the low-cost neighboring communications are used, inputs are connected to outputs of neighboring processes. In the case of such neighboring connections, the relations are local in all dimensions. It is then possible to use difference equations for our specification. Recursion relations retain more information in the sense that the "phase" of a computation wave is embedded in the description. For more complex situations, involving non-local connections, the greater generality of recursion relations is essential.

Semantics and Abstraction

By the well-known fixed-point theory[LASSEZ, NGUYEN &SONENBERG 82], the unique minimum solution of any system of recursion equations exists. This minimum solution is taken to be the function that the system computes. The process of finding this minimum solution can be described intuitively by the following successive approximation procedure. We first approximate the solution by the set of n data streams that are totally undefined in the space-time domain and substitute them into the right-hand side of the recursion equations. This substitution results in the left-hand side which is a set of data streams that are defined only on the point in space-time domain where initial values are set. These data streams are the inputs to the algorithm and we refer to them as initial streams. Now we substitute these initial streams again into the recursion equation and get another set of data streams that have even more points in the space-time domain defined. We repeat this process until no more points in the space-time domain become defined. This process corresponds exactly to the process of computation. The only restriction is that our functions must be monotonic, i.e. each data stream at any iteration is always at least as well defined as it was on the previous one. We do not allow non-monotonic functions that destroy results which have already become defined. Refer to [MANNA74] [STOY77] for the formalism.

The resulting minimum solution consists of n data streams. Each data stream is a function over $[\mathcal{V} \rightarrow \mathcal{D}]$. In order to construct a higher level system using the system we have just obtained, we need to encapsulate the system as a sequential process or a function. The function defining a sequential process or the single applicative function is from value domain to value domain. This encapsulation usually involves some transformation from the original data structure to the space-time domain for the inputs and a corresponding transformation for the outputs. Technically, the procedure is as follows:

(1) The input mapping function maps all initial/boundary values from an abstract

input data structure to the inputs unconnected to any output in the space-time domain.

(2) Compute the least fixed point solution in the space-time domain as described above.

(3) The resulting outputs occur at those points in the space-time domain designated as outputs of the system. The output mapping function maps these outputs from an element of $[\mathcal{V} \rightarrow D]^n$ to an element of the output value domain $D^{n'}$.

The result of this procedure is the abstract definition in the value domain, of the function (type $[D^{m'} \rightarrow D^{n'}]$) implemented by the space-time algorithm.

Matrix Multiplication on a Systolic Array—Program and Semantics

In his paper, Kung described various matrix related operations performed on an array of interconnected hexagonal elements. We present his algorithm for multiplying two full matrices in CRYSTAL and prove the correctness of the algorithm.

As shown in figure 1, hexagonal elements are connected into a hexagonal array. Each element has three inputs and three outputs as shown by the incoming and outgoing arrows, respectively. Such a process performs an inner product operation in the north and south direction , i.e. $c_{out} = c_{in} + a_{in} \times b_{in}$, and transmits the other two inputs as they were., i.e., $a_{out} = a_{in}$, $b_{out} = b_{in}$.

The two matrices to be multiplied, A and B, and a matrix C are fed into the array as shown in the figure. The resulting matrix C' will come out at the top of the array as shown. Kung's original algorithm assumes a global clock thus every process performs an operation synchronously. When data items are fed from the boundaries of the array, due to the fact that every process is forced to perform an operation even before any meaningful data reaches the process, proper initialization of the system by padding zeros in the input

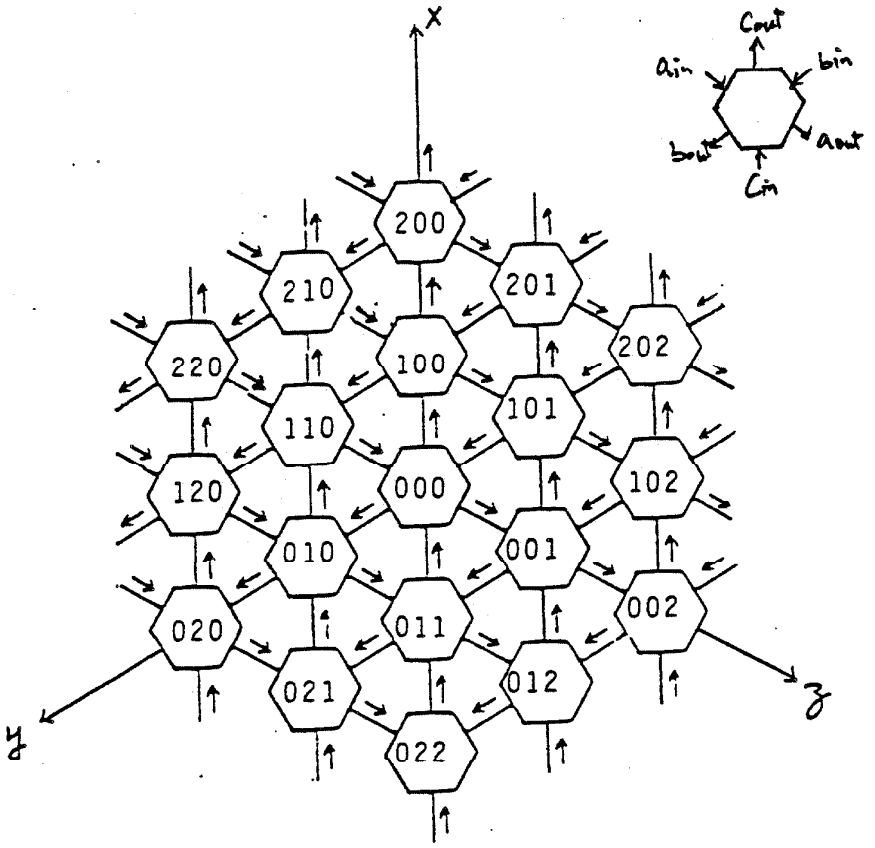

FIGURE 1. The Space Coordinate System For A Hexagonal Array

streams and disposal of garbage data are necessary. The same algorithm with a different timing scheme, e.g. self-timed [Seitz 80] scheme can simplify conceptually the interaction of processes and the flow of data and renders a simpler initiation of the system. This simplification results from the fact that the self-timed scheme assures that each process does not perform any operation until all the meaningful data items have reached the process. On the other hand, the self-timed scheme does not have any global control, the ordering of the system events is an emergent property of the local synchronizations. Thus the specification of the ordering relations among invocations of processes has to be verified from initial data arragement since self-timed elements are triggered by the arrival of data.

Both algorithms can be described by CRYSTAL programs. We will present both versions as examples of our notation and verification methodology and discuss some of the design issues of synchronous systems vs. self-timed systems.

In writing a CRYSTAL program, one need to choose an appropriate coordinate system for the processes in space. The data flow of the array has a symmetry which can be described by the dihedral group of order 3 [LIN &MEAD 82]. As shown in Figure 1, the 3-dimensional Cartesian coordinate system is chosen to reflect this symmetry. The center of this hexagon can be viewed as a corner of a cube. The hexagon is made of the three faces that contain the corner.

Next, we choose a coordinate system in the time domain according to the system timing scheme. If the system is synchronous, then t denotes the number of system clock cycles. For a self-timed system, each process has its own time frame. If it is a deterministic system, there exists a unique partial ordering of all events of the system. For a nondeterministic system, there exists more than one possible partial ordering of system events. Thus the synchronous system is a special case of deterministic systems where the unique partial ordering on events is controlled by the system clock.

Let x, y, z, t be non-negative integers. Define the following predicates which specify

the location of processes in the space-time domain as shown in Figure 2. For example φ_{xy} restricts the xy plane to an area within the specified bounds in the first quadrant.

$$
\begin{aligned}
\varphi_{xy} &\equiv (n > x > 0) \wedge (n > y > 0) \wedge (z = 0) \\
\varphi_{yz} &\equiv (n > y > 0) \wedge (n > z > 0) \wedge (x = 0) \\
\varphi_{zx} &\equiv (n > z > 0) \wedge (n > x > 0) \wedge (y = 0) \\
\varphi_{x} &\equiv (n > x > 0) \wedge (y = 0) \wedge (z = 0) \\
\varphi_{y} &\equiv (n > y > 0) \wedge (z = 0) \wedge (x = 0) \\
\varphi_{z} &\equiv (n > z > 0) \wedge (x = 0) \wedge (y = 0) \\
\varphi_{xyz} &\equiv (x = 0) \wedge (y = 0) \wedge (z = 0) \\
\varphi_{h} &\equiv \varphi_{xy} \vee \varphi_{yz} \vee \varphi_{zx} \vee \varphi_{x} \vee \varphi_{y} \vee \varphi_{z} \vee \varphi_{xyz} \\
\varphi_{a} &\equiv (0 \leq |x - y| < 3n) \wedge (0 \leq \min(2x - y, 2y - x) < 3n) \wedge (z = 0) \\
\varphi_{b} &\equiv (0 \leq |z - x| < 3n) \wedge (0 \leq \min(2z - x, 2x - z) < 3n) \wedge (y = 0) \\
\varphi_{c} &\equiv (0 \leq |y - z| < 3n) \wedge (0 \leq \min(2y - z, 2z - y) < 3n) \wedge (x = 0) \\
\varphi_{s} &\equiv \varphi_{a} \vee \varphi_{b} \vee \varphi_{c} \\
\varphi_{a'} &\equiv \varphi_{xy} \vee \varphi_{zx} \vee \varphi_{z} \\
\varphi_{b'} &\equiv \varphi_{yz} \vee \varphi_{xy} \vee \varphi_{y} \\
\varphi_{c'} &\equiv \varphi_{xy} \vee \varphi_{zx} \vee \varphi_{x} \\
\varphi_{t} &\equiv 0 \leq t \leq 4(n-1) + 1
\end{aligned}
$$

We use the notation $\overline{\varphi}$ to indicate the negation of the predicate φ. Define the space-time domain of the array $\mathcal{V} \equiv \{(x, y, z, t) : \varphi_s \wedge \varphi_t\}$. From now on unless otherwise specified, (x, y, z, t) always refers to any $(x, y, z, t) \in \mathcal{V}$.

Let A_{in}, B_{in}, and C_{in} be the input data streams (a function over $[\mathcal{V} \rightarrow D]$) and A_{out}, B_{out} and C_{out} be the output data streams. Then the following process definition specify how each output stream is related to the input streams. For example, each hexagonal element within the hexagon (when φ_h holds), has an inner product element for computing c_{out} and two delay elements for computing a_{out} and b_{out}. The definition below defines for all the elements in the space-time domain in a structured way.

Process Definition

$$
A_{out} = \lambda(x, y, z, t). \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow A_{in}(x, y, z, t-1) \\ else \rightarrow \perp \end{cases} \tag{2a}
$$

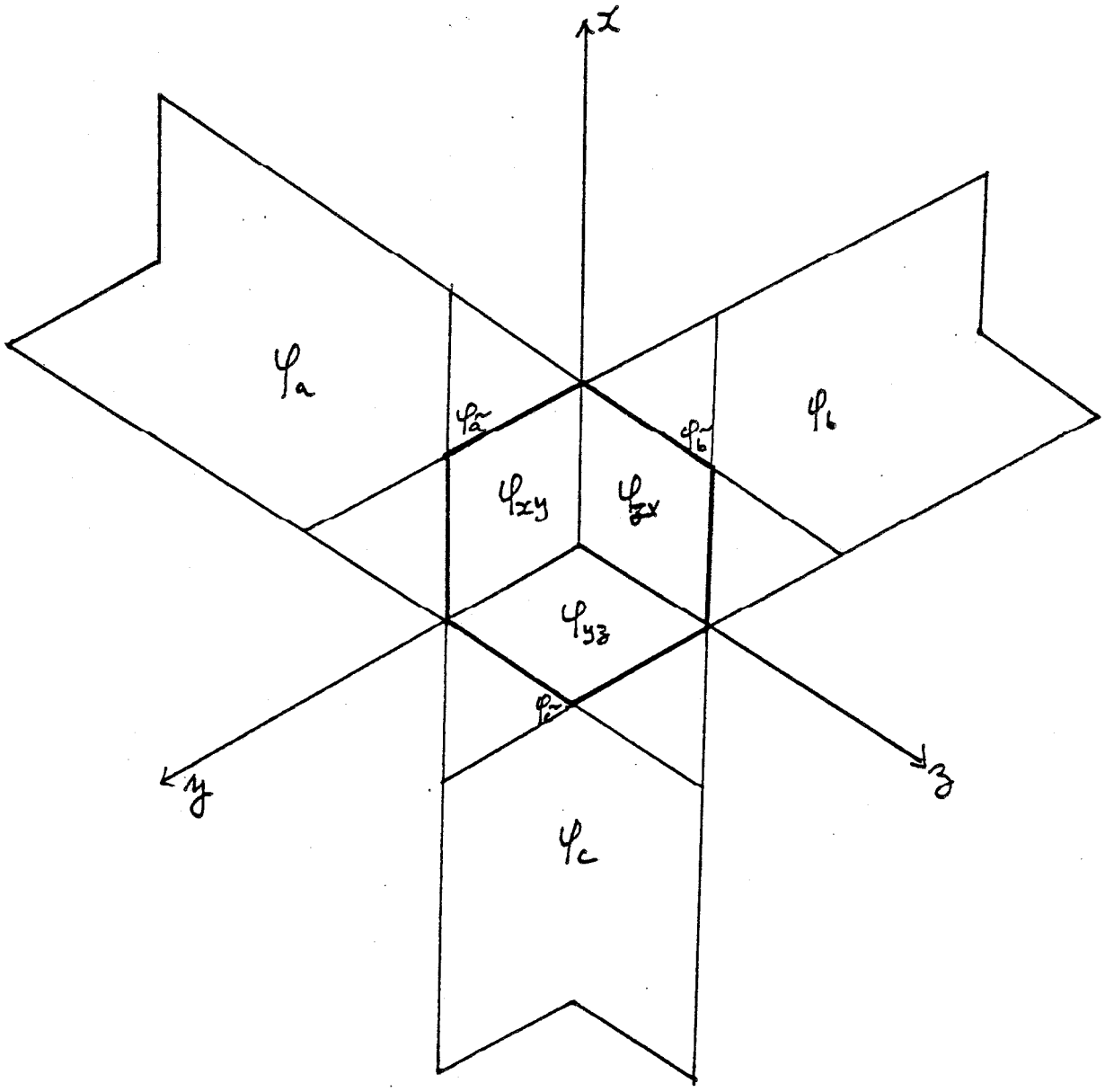
x
φ_a
$\varphi_{\tilde{a}}$
$\varphi_{\tilde{b}}$
φ_b
φ_{xy}
φ_{zx}
φ_{yz}
$\varphi_{\tilde{c}}$
y
z
φ_c

FIGURE 2

$$B_{out} = \lambda(x,y,z,t).\begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow B_{in}(x,y,z,t-1) \\ else \rightarrow \perp \end{cases} \tag{2b}$$

$$C_{out} = \lambda(x,y,z,t).\begin{cases} \varphi_c \wedge \overline{(\varphi_{yz} \vee \varphi_y \vee \varphi_z)} \rightarrow C_{in}(x,y,z,t-1) \\ \varphi_h \rightarrow C_{in}(x,y,z,t-1) + A_{in}(x,y,z,t-1) \times B_{in}(x,y,z,t-1) \\ else \rightarrow \perp \end{cases} \tag{2c}$$

Next we define the connection plans for all the elements. It specifies which output connects to which input in a structured way.

Connections

$$A_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow A_{in}(x,y,z,t) \\ t > 0 \rightarrow \begin{cases} \varphi_a \rightarrow A_{out}(x+1,y+1,z,t) \\ \varphi_{a'} \rightarrow A_{out}(x,y,z-1,t) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{3a}$$

$$B_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow B_{in}(x,y,z,t) \\ t > 0 \rightarrow \begin{cases} \varphi_b \rightarrow B_{out}(x+1,y,z+1,t) \\ \varphi_{b'} \rightarrow B_{out}(x,y-1,z,t) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{3b}$$

$$C_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow C_{in}(x,y,z,t) \\ t > 0 \rightarrow \begin{cases} \varphi_c \rightarrow C_{out}(x,y+1,z+1,t) \\ \varphi_{c'} \rightarrow C_{out}(x-1,y,z,t) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{3c}$$

By substituting (3a) into (2a), we obtain

$$A_{out} = \lambda(x,y,z,t).\begin{cases} t = 1 \rightarrow \begin{cases} \varphi_a \rightarrow A_{in}(x,y,z,0) \\ \varphi_{a'} \rightarrow 0 \end{cases} \\ t > 1 \rightarrow \begin{cases} \varphi_a \rightarrow A_{out}(x+1,y+1,z,t-1) \\ \varphi_{a'} \rightarrow A_{out}(x,y,z,t-1) \end{cases} \end{cases} \tag{4}$$

Similarly, we can substitute (3b) into (2b) and (3a), (3b), (3c) into (2c) to obtain a system of recursion equations in A_{out}, B_{out} and C_{out} and the initial conditions.

These equations define the behavior of the hexagonal array itself independent of the input to the array. Next, we specify the input and output transformation functions which relate the structure of the hexagonal array with the abstract data structure of matrices.

Let $h^1 = (h_a^1, h_b^1, h_c^1)$ denote the initial streams and $h^\infty = (h_a^\infty, h_b^\infty, h_c^\infty)$ denote the final streams which are the minimum solution of the above system of recursion equations (4). A Matrix with elements from the domain $\mathcal{D}$ can be thought of as a function from the domain $\mathcal{N}^2$ of index pairs to $\mathcal{D}$. We denote the resulting matrix by A', B' and C' and define the following domain,

$$\begin{aligned} \text{domain of integers from 0 to } n-1\text{: } & \mathcal{N} \\ \text{domain of matrices: } & \mathcal{M} \equiv [\mathcal{N}^2 \to \mathcal{D}], \\ \text{domain of data streams: } & \mathcal{S} \equiv [\mathcal{V} \to \mathcal{D}], \\ \text{domain of transformation functions: } & \mathcal{T} \equiv [\mathcal{V} \to \mathcal{N}^2] \\ & \mathcal{T}' \equiv [\mathcal{N}^2 \to \mathcal{V}] \end{aligned} \tag{5}$$

We define the input transformation function $(g_a, g_b, g_c) \in \mathcal{T}^3$:

$$g_a \equiv (I_a, J_a), \quad g_b \equiv (I_b, J_b), \quad g_c \equiv (I_c, J_c)$$

where

$$I_a \equiv \lambda(x,y,z,t).\begin{cases} 2y - x \equiv 0 \pmod 3 \to \frac{2y-x}{3} \\ else \to \perp \end{cases}$$

$$J_a \equiv \lambda(x,y,z,t).\begin{cases} 2x - y \equiv 0 \pmod 3 \to \frac{2x-y}{3} \\ else \to \perp \end{cases} \tag{6a}$$

$$I_b \equiv \lambda(x,y,z,t).\begin{cases} 2x - z \equiv 0 \pmod 3 \to \frac{2x-z}{3} \\ else \to \perp \end{cases}$$

$$J_b \equiv \lambda(x,y,z,t).\begin{cases} 2z - x \equiv 0 \pmod 3 \to \frac{2z-x}{3} \\ else \to \perp \end{cases} \tag{6b}$$

$$I_c \equiv \lambda(x,y,z,t).\begin{cases} 2y - z \equiv 0 \pmod 3 \to \frac{2y-z}{3} \\ else \to \perp \end{cases}$$

$$J_c \equiv \lambda(x,y,z,t).\begin{cases} 2z-y \equiv 0 \pmod 3 \rightarrow \frac{2z-y}{3} \\ else \rightarrow \perp \end{cases} \tag{6c}$$

We use the shorthand notation

$$g_a(x,y,z,t) \equiv (I_a(x,y,z,t), J_a(x,y,z,t))$$

for component-wise application of the argument (x,y,z,t) to a vector of functions such as (I_a, J_a).

then the inital streams are defined as

$$h_a^1 = \lambda(x,y,z,t).\begin{cases} t=0 \rightarrow \begin{cases} \varphi_a \rightarrow \begin{cases} (2y-x \equiv 0 \pmod 3) \wedge (2x-y \equiv 0 \pmod 3) \\ \qquad \rightarrow Ag_a(x,y,z,t) \\ else \rightarrow 0 \end{cases} \\ \varphi_{a'} \rightarrow 0 \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases}$$

$$h_b^1 = \lambda(x,y,z,t).\begin{cases} t=0 \rightarrow \begin{cases} \varphi_b \rightarrow \begin{cases} (2x-z \equiv 0 \pmod 3) \wedge (2z-x \equiv 0 \pmod 3) \\ \qquad \rightarrow Bg_b(x,y,z,t) \\ else \rightarrow 0 \end{cases} \\ \varphi_{b'} \rightarrow 0 \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases} \tag{7}$$

$$h_c^1 = \lambda(x,y,z,t).\begin{cases} t=0 \rightarrow \begin{cases} \varphi_c \rightarrow \begin{cases} (2y-z \equiv 0 \pmod 3) \wedge (2z-y \equiv 0 \pmod 3) \\ \qquad \rightarrow Cg_c(x,y,z,t) \\ else \rightarrow 0 \end{cases} \\ \varphi_{c'} \rightarrow 0 \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases}$$

where $h_a^1, h_b^1, h_c^1 \in S, \qquad A,B,C \in \mathcal{M}, (*in)$

Output Transformation Function $(g'_a, g'_b, g'_c) \in \mathcal{T}'^3$:

$$g'_a \equiv (X_a, Y_a, Z_a, T_a), \quad g'_b \equiv (X_b, Y_b, Z_b, T_b), \quad g'_c \equiv (X_c, Y_c, Z_c, T_c).$$

where

$$
\begin{aligned}
&X_a \equiv \lambda(i,j).\max(j-i,0),\\
&Y_a \equiv \lambda(i,j).\max(i-j,0),\\
&Z_a \equiv \lambda(i,j).n-1.\\
&X_b \equiv \lambda(i,j).\max(i-j,0),\\
&Y_b \equiv \lambda(i,j).n-1,\\
&Z_b \equiv \lambda(i,j).\max(j-i,0).\\
&X_c \equiv \lambda(i,j).n-1,\\
&Y_c \equiv \lambda(i,j).\max(i-j,0),\\
&Z_c \equiv \lambda(i,j).\max(j-i,0).\\
&T_a \equiv T_b \equiv T_c \equiv \lambda(i,j).n+min(i,j)+i+j. \qquad (8)
\end{aligned}
$$

As before, $g'_a(i,j)$ denotes the component-wise application of (i,j) to the four components of g'_a.

The resulting matrices are defined as follows,

$$A' = h_a^{\infty} g'_a, B' = h_b^{\infty} g'_b, C' = h_c^{\infty} g'_c,$$

where $h_a^{\infty}, h_b^{\infty}, h_c^{\infty} \in S$, the final streams

$$A', B', C' \in M; \qquad g'_a, g'_b, g'_c \in \mathcal{T}'(*out)$$

We now verify that the above system of recursion equations and the input output transformation functions correctly implement the familiar matrix operations, i.e.

$$A'(i,j) = A(i,j) \qquad (9a)$$

$$B'(i,j) = B(i,j) \qquad (9b)$$

$$C'(i,j) = \sum_{k=0}^{n-1} A(i,k) \times B(k,j) + C(i,j) \qquad (9c)$$

where $0 \leq i < n, \quad 0 \leq j < n$

We verify first the following lemmas which are the final streams (the solution of the recursion equations) in the space-time domain.

Lemma A,B:

$$A_{out}(x,y,z,t) = \begin{cases} \varphi_a \vee \varphi_{a'} \rightarrow \\ \quad A_{in}(x+\max(t-1-z,0), y+\max(t-1-z,0), \max(z-(t-1),0), 0) \\ else \rightarrow \perp \end{cases} \tag{10a}$$

$$B_{out}(x,y,z,t) = \begin{cases} \varphi_b \vee \varphi_{b'} \rightarrow \\ \quad B_{in}(x+\max(t-1-y,0), \max(y-(t-1),0), z+\max(t-1-y), 0) \\ else \rightarrow \perp \end{cases} \tag{10b}$$

Lemma C:

Let $U_1 \equiv t-1+k-y$, $U_2 \equiv t-1+k-z$, $V_1 \equiv (t-1-x-k)-(y+k)$, $V_2 \equiv (t-1-x-k)-(z+k)$, $K_1 \equiv 1-\min(x,t-1)$ and $K_2 \equiv \min(n-1-y, n-1-z, t-1-x)$. Define

$$S_1 \equiv \sum_{k=K_1}^{0} A_{in}(x+k+\max(U_2,0), y+\max(U_2,0), \max(-U_2,0), 0) \times B_{in}(x+k+\max(U_1,0), \max(-U_1,0), z+\max(U_1,0), 0)$$

and

$$S_2 \equiv \sum_{k=0}^{K_2} A_{in}(\max(V_2,0), y+k+\max(V_2,0), \max(-V_2,0), 0) \times B_{in}(\max(V_1,0), \max(-V_1,0), z+k+\max(V_1,0), 0),$$

then

$$C_{out}(x,y,z,t) = \begin{cases} \varphi_c \vee \varphi_{c'} \rightarrow \\ \quad C_{in}(\max(x-(t-1),0), y+\max(t-1-x,0), z+\max(t-1-x,0), 0) \\ \quad\quad +\lambda(x,y,z,t).S_1(x,y,z,t) + \lambda(x,y,z,t).S_2(x,y,z,t) \\ else \rightarrow \perp \end{cases} \tag{10c}$$

These relate outputs on any point in space-time domain to the initial input streams. Since they are total functions in space-time domain, the solution of the equations is automatically the minimum. Thus one way of verify them is simply substitute them into

the recursion equation and check if the equations hold. The simple substitution technique will not work in general, since the final streams are not necessarily total in the space-time domain. In this case, an inductive proof showing that the final streams are the minimum solution is necessary.

The computation waves of such a system are very instructive in such proofs. We observe that there are two triangular waves, one incident wave proceeding toward the origin. Another is a reflected wave proceeding outward from the origin. We define the phase of the reflected wave to be $x + y + z - t$ and that of the incident wave to be $x + y + z + 2t$. A wave front is defined in the traditional way as the locus of all points of the wave having the same phase. With t fixed, there are many of such wavefronts spread out in space. In this particular system, we number these wavefront positions by $w = x + y + z$. Notice that the partial result of a particular element of a matrix is carried on by a single wavefront with one value of phase. Intuitively, the induction for the reflected wave is on the wavefront of phase ϕ and $w = k$ to phase ϕ and $w = k + 1$. For the incident wave, inductions proceeds from $w = k$ to $w = k - 2$.

The pair (ϕ, w) can be formalized as a well-founded set (a set with no infinite decreasing sequences, so that induction is valid on the set) with the binary relation $\prec$, which is defined as the transitive closure of $\prec^*$, the binary relation defined below only on neighboring elements:

$$\begin{aligned} &\text{incident wave: } (\phi_1, w_1) \prec^* (\phi_2, w_2) \text{ if } \phi_1 = \phi_2 \text{ and } w_1 = w_2 + 2. \\ &\text{reflected wave: } (\phi_1, w_1) \prec^* (\phi_2, w_2) \text{ if } \phi_1 = \phi_2 \text{ and } w_1 = w_2 - 1. \end{aligned} \tag{11}$$

The inductive proof is given in [CHEN 82].

By composing the input and output transformation functions with the initial and final streams, we obtain (9a), (9b), and (9c). The detailed proof is in [CHEN 82].

Self-timed Systems

A self-timed system differs from a synchronous system in the sequencing of system events. In a synchronous system, all processes are activated simultaneously by the same

clock cycle, i.e. all processes have the same invocation number. The invocations of each process are ordered by the linear sequence of the clock. Thus all the invocations (s,t), where s is the space parameter and t is the time parameter, have a unique partial ordering $\prec$ defined to be

$$(s_1,t_1) \prec (s_2,t_2) \text{ if } s_1 = s_2 \text{ and } t_1 < t_2.$$

In a self-timed system, the ordering of the system events is not self-evident as in the synchronous system. Each process is invoked only when all of its inputs are ready. Thus the overall system timing is an emergent property of the ensemble attributed by the local synchronization of all the processes in the system. In such a system, the "time" component of the space-time domain is a function of the space component, i.e., each process has its own time-frame. The relation between time-coordinates of two communicating processes needs to be asserted in the connection plans. This time-domain relationship among processes must be verified since it depends on the initial data arrangement because self-timed elements are triggered by the input data.

The following is the space-time algorithm for the self-timed matrix multiplication on the systolic array. We define a few more predicates to specify where the initial data will be put. Notice that this set of predicates covers much less area than the set φ_a, φ_b and φ_c, since the self-timed algorithm does not need padding zeros in the input data streams.

$$\begin{aligned}
\varphi_{\tilde{a}} \equiv& (0 \le |x-y| < n) \wedge (0 \le \min(x,y) < n) \wedge (z = 0) \\
\varphi_{\tilde{b}} \equiv& (0 \le |z-x| < n) \wedge (0 \le \min(z,x) < n) \wedge (y = 0) \\
\varphi_{\tilde{c}} \equiv& (0 \le |y-z| < n) \wedge (0 \le \min(y,z) < n) \wedge (x = 0)
\end{aligned}$$

We also redefine

$$\begin{aligned}
\varphi_s \equiv& \varphi_{\tilde{a}} \vee \varphi_{\tilde{b}} \vee \varphi_{\tilde{c}} \\
\varphi_t(x,y,z) \equiv& 0 \le t \le n-1-\max(x,y,z)
\end{aligned}$$

Process Definition

$$A_{out} = \lambda(x,y,z,t).\begin{cases} \varphi_{\tilde{a}} \vee \varphi_{a'} \rightarrow A_{in}(x,y,z,t(x,y,z)) \\ else \rightarrow \perp \end{cases} \qquad (12a)$$

$$B_{out} = \lambda(x,y,z,t).\begin{cases} \varphi_{\tilde{b}} \vee \varphi_{b'} \rightarrow B_{in}(x,y,z,t(x,y,z)) \\ else \rightarrow \perp \end{cases} \tag{12b}$$

$$C_{out} = \lambda(x,y,z,t).\begin{cases} \varphi_{\tilde{c}} \wedge \overline{(\varphi_{yz} \vee \varphi_y \vee \varphi_z)} \rightarrow C_{in}(x,y,z,t(x,y,z)) \\ \varphi_h \rightarrow C_{in}(x,y,z,t(x,y,z)) + A_{in}(x,y,z,t(x,y,z)) \times B_{in}(x,y,z,t(x,y,z)) \\ else \rightarrow \perp \end{cases} \tag{12c}$$

Connection Plans

$$A_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow A_{in}(x,y,z,t(x,y,z)) \\ t > 0 \rightarrow \begin{cases} \varphi_{\tilde{a}} \rightarrow A_{out}(x+1,y+1,z,t(x,y,z)-1) \\ \varphi_{a'} \rightarrow A_{out}(x,y,z-1,t(x,t,z)) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{13a}$$

$$B_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow B_{in}(x,y,z,t(x,y,z)) \\ t > 0 \rightarrow \begin{cases} \varphi_{\tilde{b}} \rightarrow B_{out}(x+1,y,z+1,t(x,y,z)-1) \\ \varphi_b \rightarrow B_{out}(x,y-1,z,t(x,y,z)) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{13b}$$

$$C_{in} = \lambda(x,y,z,t).\begin{cases} t = 0 \rightarrow C_{in}(x,y,z,t(x,y,z)) \\ t > 0 \rightarrow \begin{cases} \varphi_{\tilde{c}} \rightarrow C_{out}(x,y+1,z+1,t(x,y,z)-1) \\ \varphi_{c'} \rightarrow C_{out}(x-1,y,z,t(x,y,z)) \end{cases} \\ else \rightarrow \perp. \end{cases} \tag{13c}$$

Input transformation functions: The functions $(g_a, g_b, g_c) \in \mathcal{T}^3$ map from the space-time domain $\mathcal{V}$ to the matrix indices $\mathcal{N}^2$ as specified in (5).

$$g_a \equiv (\lambda(x,y,z,t).y, \lambda(x,y,z,t).x)$$

$$g_b \equiv (\lambda(x,y,z,t).x, \lambda(x,y,z,t).z)$$

$$g_c \equiv (\lambda(x,y,z,t).y, \lambda(x,y,z,t).z)$$

The initial streams are defined by using the composition of the input transformation functions and the matrix function.

$$h_a^1 = \lambda(x,y,z,t)\begin{cases} t=0 \rightarrow \begin{cases} \varphi_{\tilde{a}} \rightarrow A g_a(x,y,z,t) \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases}$$

$$h_b^1 = \lambda(x,y,z,t)\begin{cases} t=0 \rightarrow \begin{cases} \varphi_{\tilde{b}} \rightarrow B g_b(x,y,z,t) \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases}$$

$$h_c^1 = \lambda(x,y,z,t)\begin{cases} t=0 \rightarrow \begin{cases} \varphi_{\tilde{c}} \rightarrow C g_c(x,y,z,t) \\ else \rightarrow \perp \end{cases} \\ t>0 \rightarrow \perp \end{cases} \tag{14}$$

Output Transformation Functions: The functions $(g'_a, g'_b, g'_c) \in \mathcal{T}'^3$ define the space-time coordinates associated with each element of the output matrix.

$$g'_a \equiv (X_a, Y_a, Z_a, T_a), \quad g'_b \equiv (X_b, Y_b, Z_b, T_b), \quad g'_c \equiv (X_c, Y_c, Z_c, T_c).$$

where

$$\begin{aligned} X_a &\equiv \lambda(i,j).j - \min(i,j), \\ Y_a &\equiv \lambda(i,j).i - \min(i,j), \\ Z_a &\equiv \lambda(i,j).n-1-\min(i,j). \\ X_b &\equiv \lambda(i,j).i - \min(i,j), \\ Y_b &\equiv \lambda(i,j).n-1-\min(i,j), \\ Z_b &\equiv \lambda(i,j).j - \min(i,j). \\ X_c &\equiv \lambda(i,j).n-1-\min(i,j), \\ Y_c &\equiv \lambda(i,j).i - \min(i,j), \\ Z_c &\equiv \lambda(i,j).j - \min(i,j). \\ T_a &\equiv T_b \equiv T_c \equiv \lambda(i,j).\min(i,j). \end{aligned} \tag{15}$$

In the connection plan we have used the following assertions.

$$t(x,y,z) = \begin{cases} \varphi_{\tilde{a}} \rightarrow t(x+1,y+1,z)+1 \\ \varphi_{a'} \rightarrow t(x,y,z-1) \\ \varphi_{\tilde{b}} \rightarrow t(x+1,y,z+1)+1 \\ \varphi_{b'} \rightarrow t(x,y-1,z) \\ \varphi_{\tilde{c}} \rightarrow t(x,y+1,z+1)+1 \\ \varphi_{c'} \rightarrow t(x-1,y,z) \end{cases} \tag{16}$$

Proof :

We prove these assertions by induction on the well-founded set of wavefront number k.

$$K = \{\, k = \frac{x+y+z}{3} + t(x,y,z) : x,y,z \text{ and } t \text{ are non-negative integers.} \}$$

with the usual less-than ($<$) ordering on the rationals.

(i) base case: $k = 0$. Since x, y, z, t are all non-negative integers, we have $x = y = z = t = 0$. By (14), $A_{in}(0,0,0,0)$, $B_{in}(0,0,0,0)$ and $C_{in}(0,0,0,0)$ are initial data, thus the process is initiated. Notice also that none of the other processes can proceed since there is at least one input undefined for each of them. The induction hypothesis is vacuously true in this case.

(ii) induction step. We assume that the hypothesis (16) is true for all $k < k_0$. We then show that it must be true for $k = k_0$. The proof consists of three steps.

(1) For a given k, inputs to processes of invocation $t(x,y,z) - 1$ were generated by processes with $t(x,y,z) - 1$ or $t(x,y,z) - 2$. We use this fact later to demonstrate that the processes' invocations occur in lock step, i.e. no process invoked more frequently than any other.

(2) All outputs of the previous invocation have been taken before another invocation is initiated.

(3) All inputs to an invocation $t(x,y,z)$ come from invocations of $t(x,y,z)$ and $t(x,y,z)-1$, depending upon location.

We consider a process at (x,y,z) with $k_0 = \frac{x+y+z}{3} + t(x,y,z)$. From Figure 1, it has three inputs *a.in*, *b.in* and *c.i* from the following neighboring processes respectively.

$$a_{in}: \begin{cases} \varphi_{\bar{a}} \to (x+1, y+1, z) \\ \varphi_{a'} \to (x, y, z-1) \end{cases}$$

$$b_{in}: \begin{cases} \varphi_{\bar{b}} \to (x+1, y, z+1) \\ \varphi_{b'} \to (x, y-1, z) \end{cases}$$

$$c_{in}: \begin{cases} \varphi_{\bar{c}} \to (x, y+1, z+1) \\ \varphi_{c'} \to (x-1, y, z) \end{cases} \tag{17}$$

Since $k_0 - 1 < k_0$, the hypothesis is true for this same process at $t(x,y,z)-1$, the previous invocation. By the induction hypothesis (16), this invocation takes its inputs from the above processes at their time $t(x,y,z)-1$ or $t(x,y,z)-2$ depending on where the process is located.

This process provides outputs to the following neighboring processes.

$$a_{out}: \begin{cases} \varphi_{\bar{a}} \wedge (x>0) \wedge (y>0) \to (x-1, y-1, z) \\ \varphi_{a'} \vee \varphi_x \vee \varphi_y \to (x, y, z+1) \end{cases}$$

$$b_{out}: \begin{cases} \varphi_{\bar{b}} \wedge (x>0) \wedge (z>0) \to (x-1, y, z-1) \\ \varphi_{b'} \vee \varphi_x \vee \varphi_z \to (x, y+1, z) \end{cases}$$

$$c_{out}: \begin{cases} \varphi_{\bar{c}} \wedge (z>0) \wedge (y>0) \to (x, y-1, z-1) \\ \varphi_{c'} \vee \varphi_z \vee \varphi_y \to (x+1, y, z) \end{cases}$$

We assert that these outputs from the invocation number $t(x,y,z)-1$ of process (x,y,z) are taken by either invocation number $(t(x,y,z)-1)$ or $t(x,y,z)$ of these neighboring process depending upon their locations. Since

$$\frac{x+y+z+1}{3} + (t(x,y,z)-1) = \frac{x-1+y-1+z}{3} + (t(x,y,z)-1) + 1 = k - \frac{2}{3} < k$$

Thus the induction hypothesis can be applied. Process (x, y, z) is ready to start a new invocation once all of its three inputs are ready. Since the processes that provide output to it at their respective time frame $t(x, y, z) - 1$ or $t(x, y, z)$ satisfy the following inequality

$$\frac{x+1+y+1+z}{3} + t(x, y, z) - 1 = \frac{x+y+z-1}{3} + t(x, y, z) = k - \frac{1}{3} < k,$$

the induction hypothesis applies to them. Process (x, y, z) has three and only three inputs ready at their respective locations after its invocation number $t(x, y, z) - 1$. The align-element insures that no invocation can occur before all three inputs are ready, thus process (x, y, z) has its invocation number $t(x, y, z)$ occurring. This proves the above assertions.□

This algorithm is also deterministic for we can define a partial ordering $\prec$ on the invocations of processes. This binary relation is defined as the transitive closure of the binary relation $\prec^*$ as follows.

$$(x, y_1, z, t(x, y, z)) \prec^* (x_0, y_0, z_0, t(x_0, y_0, z_0))$$

If there exist $(x_1, y_1, z_1, t(x_1, y_1, z_1))$,$(x_2, y_2, z_2, t(x_2, y_2, z_2))$, and $(x_3, y_3, z_3, t(x_3, y_3, z_3))$ such that

$$\begin{cases} \varphi_{\tilde{a}} \rightarrow ((x_1 = x_0) \wedge (y_1 = y_0) \wedge (z_1 = z_0 - 1) \wedge (t(x_1, y_1, z_1) = t(x_0, y_0, z_0))) \\ \varphi_{a'} \rightarrow ((x_1 = x_0 + 1) \wedge (y_1 = y_0 + 1) \wedge (z_1 = z_0) \wedge (t(x_1, y_1, z_1) = t(x_0, y_0, z_0) - 1)) \end{cases} \tag{18a}$$

and

$$\begin{cases} \varphi_{\tilde{b}} \rightarrow ((x_2 = x_0) \wedge (y_2 = y_0 - 1) \wedge (z_2 = z_0) \wedge (t(x_2, y_2, z_2) = t(x_0, y_0, z_0))) \\ \varphi_{b'} \rightarrow ((x_2 = x_0 + 1) \wedge (y_2 = y_0) \wedge (z_2 = z_0 + 1) \wedge (t(x_2, y_2, z_2) = t(x_0, y_0, z_0) - 1)) \end{cases} \tag{18b}$$

and

$$\begin{cases} \varphi_{\tilde{c}} \rightarrow (x_3 = x_0 - 1) \wedge (y_3 = y_0) \wedge (z_3 = z_0) \wedge (t(x_3, y_3, z_3) = t(x_0, y_0, z_0)) \\ \varphi_{c'} \rightarrow (x_3 = x_0) \wedge (y_3 = y_0 + 1) \wedge (z_3 = z_0 + 1) \wedge (t(x_3, y_3, z_3) = t(x_0, y_0, z_0) - 1) \end{cases} \tag{18c}$$

This definition can be derived from (16) with the existance of the align-element for the inputs of each process as an assumption.

Now we proceed to verify the algorithm by proving two lemmas.

Lemma a,b:

$$A_{out}(x,y,z,t) \equiv \begin{cases} \varphi_{\bar{a}} \vee \varphi_{a'} \rightarrow \\ \quad A_{in}(x+t, y+t, 0, 0) \\ else \rightarrow \perp \end{cases} \tag{19a}$$

$$B_{out}(x,y,z,t) \equiv \begin{cases} \varphi_{b} \vee \varphi_{b'} \rightarrow \\ \quad B_{in}(x+t, 0, z+t, 0) \\ else \rightarrow \perp \end{cases} \tag{19b}$$

Lemma C:

$$C_{out}(x,y,z,t) = \begin{cases} \varphi_{\bar{c}} \vee \varphi_{c'} \rightarrow C_{in}(\max(x-t,0), y+\max(t-x,0), z+\max(t-x,0), 0) \\ \quad + \sum_{k=0}^{x+t} A_{in}(k, y+t, 0, 0) \times B_{in}(k, 0, z+t, 0) \\ else \rightarrow \perp \end{cases} \tag{19c}$$

Similar to the synchronous case, we can either prove by direct substitution or by induction on K, the set of wavefront number. By composing these lemmas with input and output transformation functions, we can derive (9a), (9b) and (9c).

From both algorithms, we observe that the input and output transformation functions and the semantics of the hexagonal array are much simpler for the self-timed version. This result is not accidental, for the interaction among flows of data for this particular algorithm only utilizes one third of the maximum space-time resources. In the self-timed version, only one third of the processes (all processes with the same $k = \frac{x+y+z}{3} + t(x,y,z)$) are active at any instant. In the synchronous version, all processes are active at all times, thus padding zeros are necessary. The simplicity of the self-timed version is a pay-off of the more sophisticated synchronization method. It is necessary to prove that local synchronization gives rise to the global sequencing relations among all the processes. Describing these two algorithms in CRYSTAL not only shows the capability of our framework but also provides

many insights into the the complexity of various aspects of these two different timing schemes. We have achieved one of the important goals of this research – by providing a formalism in which one can gain a much deeper understanding of the subject one describes in the process of so doing.

Conclusion

We have presented a notation and formal semantics for general non-linear systems with memory. An essential part of the semantics is a methodology for abstracting the behavior of such systems so they can be used as components at a higher level. The semantics of a particular system consists of

(i) An input mapping function from the value domain to the space-time structure of the system.

(ii) A function in the space-time domain which completely defines the operation of the system.

(iii) An output mapping function from the space-time structure to the value domain.

The abstract semantics of the system is obtained by eliminating space-time variables to yield a function in the value domain alone. When it is possible to eliminate all intermediate variables, as it was with the Kung array, the abstract system is purely functional. When some intermediate state variables remain, as in the case where real-time input is necessary, the system is defined by an abstract sequential process. Such a process is defined by a system of recursion relations in time. From an engineering point of view, the input and output mapping functions serve as precise interface specifications for the system.

The methodology can be applied to any system: linear, non-linear, time-varying, history-dependent. We believe it provides, for the first time, a unified approach spaning the range from computer programs to linear transfer functions; from transistor circuits to high level communicating sequential processes.

References

[ACKERMAN 82] Ackerman, W.B.
Data Flow Languages
Computer, 15(2):15-26, February 1982.

[BACKUS 78] Backus, J. *Can Programming Be Liberated from the von Neumann Style?*
A Functional Style and Its Algebra of Programs.
CACM, 21(8)613-641, August 1978.

[BROWNING 80] Browning, S.A.
The Tree Machine: A Highly Concurrent Computing Environment

Ph.D. thesis, California Institute of Technology, January, 1980.

[CHEN 82] Chen, M.C.
Ph.D.Thesis in Preparation, California Institute of Technology.

[DIJKSTRA 76] Dijkstra, E.W.
A Discipline of Programming.
Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[JOHNSSON 81] Johnsson, L., Weiser, U., Cohen, D. and Davis, A.
Towards a Formal Treatment of VLSI Arrays
Technical Report 4191, California Institute of Technology, January, 1981.

[KAHN 74] Kahn, G.
Proc. IFIP Congress 74.
The Semantics of a Simple Language for Parallel Programming, 1974.

[KUNG &LEISERSON 80]
Kung,H.T. and Leiserson C.E.
Algorithms for VLSI Processor Arrays.
Mead &Conway, Introduction to VLSI Systems Addison-Wesley, 1980, chapter 8.3.

[KUNG, S.Y. 80] Kung S. Y.
VLSI Array Processor for Signal Processing.

Conference on Advanced Research in Integrated Circuits, MIT., 1980.

[LASSEZ,NGUYEN&SONENBERG]
Lassez,J.L., Nguyen,V.L, and Sonenberg,E.A.
Fixed Point Theorems and Semantics: A Folk Tale
Information Processing Letters, 14(3):112-116, February 1982.

[LIN &MEAD]
Lin T.Z. and Mead C.A.
The Application of Group Theory in Classifying Systolic Arrays
Display File 5006, California Institute of Technology, March, 1982.

[MANNA 74]
Manna, Z.
Mathematical Theory of Computation.
McGraw-Hill, New York, 1974.

[PERLIS 82]
Perlis, A.J.
Epigrams on Programming.
SIGPLAN Notices, 17(9), September 1982

[SEITZ 80]
Seitz, C.
System Timing.
Mead &Conway, Introduction to VLSI Systems Addison-Wesley, 1980, chapter 8.3.

[SCOTT & STRACHEY 71]
Scott, D. and Strachey, C.
Toward a Mathematical Semantics for Computer Languages
Fox, J., editor. Polytechnic institute of Brooklyn Press, New York, 1971.

[STOY 77]
Stoy, J.E.
Denotational Semantics: The Scott-Strachey Approach.
The MIT Press, Cambridge, Massachusetts, 1977.

[SUTHERLAND & MEAD 77]
Sutherland I. and Mead C.
Micro-electronics and Computer Science
Scientific American 237(3):210-229, September, 1977.